John von Neumann Institute for Computing

Tools for Parallel Quantum Chemistry Software

Thomas Steinke

published in

Modern Methods and Algorithms of Quantum Chemistry,
Proceedings, Second Edition, J. Grotendorst (Ed.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **3**, ISBN 3-00-005834-6, pp. 67-96, 2000.

http://www.fz-juelich.de/nic-series/

TOOLS FOR PARALLEL QUANTUM CHEMISTRY SOFTWARE

THOMAS STEINKE
Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)
Takustr. 7
14195 Berlin-Dahlem
Germany
E-mail: steinke@zib.de

The lack of appropriate middleware for implementing parallel methods on true parallel computers like the Intel series in late 80s forced the design of suitable programming tools for application packages. The TCGMSG was one of the first examples which provides a portable and efficient message passing interface. This tool is still used in production codes in the field of quantum chemistry. Further developments led to programming interfaces (e.g. Global Array Toolkit) providing a shared memory programming model on distributed memory architectures. This article gives an introduction into the basic terminology and the application of these tools.

1 Introduction

First-principle quantum-chemical simulations of *large* molecular systems consisting of a few hundreds of atoms are going to be more and more feasible as common "daily" task as powerful parallel processor computers are becoming available. Massively-parallel processor platforms (e.g. CRAY T3E, IBM SP) as well as large scalable shared memory systems (e.g. SGI Origin) both providing some tens or hundreds of GigaFlop/s sustained performance are accessible for scientists at certain sites. Today, a scientist who is developing parallel applications for high-performance computer platforms can choose between some programming libraries, programming models and toolkits. But, the situation was quite different in the late 80s.

In order to take advantage of new hardware features provided by the first true parallel computers 15 years ago (e.g. Intel Hypercube, later Paragon etc.), the redesign of serial versions of existing quantum chemistry packages (e.g. COLUMBUS, GAMESS (US), GAMESS-UK, DGauss) was initiated as well as the new design and implementation of packages from scratch (e.g. NWChem, Q-Chem). Due to the lack of appropriate middleware for implementing software on such distributed memory architectures research groups in the chemistry community developed suitable communication libraries and user-friendly APIs for programming these parallel machines. For example, TCGMSG was developed to provide a portable communication library for quantum chemistry software. Although the functionality of TCGMSG is superceeded by standard messsage-passing libraries PVM and MPI this software is still in use in production codes.

This article gives an introduction into the fundamentals of some of the programming tools used in production codes for quantum chemical simulations. The part of informations provided are restricted to the basics of the topic. A more complete view of the capabilities of tools explained below will arise together with the overhead presentation given at the Winterschool at Forschungszentrum Jülich.

2 Basic Tasks in Typical Quantum Chemical Calculations

The major part of today's quantum chemical calculations carried out routinely is dealing with Hartree-Fock SCF, MP2, or DFT type calculations of more or less large molecules. To identify the computational steps which take advantage of parallelization we will give a coarse-grain analysis of such calculations. We will focus on common SCF type calculations for solving the stationary Schrödinger equation $\hat{H}\Psi = E\Psi$ within the common molecular orbital (MO) approach. The following summary lists the basic time consuming steps:

1. the computation of 2-electron integrals (ERI),
2. the construction of the Fock matrix **F** from ERIs and density matrix **P**,
3. the diagonalization of **F** delivering MO coeffients **C**, and
4. the calculation of a new density matrix **P** from **C**

In *large* molecular systems consisting of a few hundreds of atoms, the number of basis functions included is in the order of thousands. In the *direct* SCF scheme the steps 1 and 2 are combined with sophisticated integral screening techniques to reduce the number of ERI contributions to be considered. In this way, any storing of ERIs either on disk (conventional approach) or in memory (in-core scheme) is avoided by (re-)computing ERIs for the Fock matrix build as they needed.

From the picture given above one can imagine that

- the *work* distribution to calculate ERIs and to construct the Fock matrix **F** which itself should be distributed over all compute nodes,
- the diagonalization of **F**, or quasi-Newton orbital rotations, and finally
- subsequent operations on the wavefunction (**C** and/or **P**) like matrix operations

should be the subject of parallelization efforts. In the next subsection some basic issues related to parallel programming are discussed.

2.1 Issues related to parallel programming

Given the list of basic tasks above one can define essential features of a suitable toolkit required for quantum-chemical calculations. Below some key issues regarding support for parallelism, not only in the context of quantum chemistry, are explained:

Work sharing, task parallelism: That is the basic idea of parallelism. It is wanted to use as many compute nodes as possible without substantial overhead in order to reduce the wall-time for a given problem size. A given set of tasks, e.g. the calculation of ERIs or elements of **F**, should be distributed over all computing nodes.

Work sharing within an application may be at a high level (*coarse-grain parallelism*) or low level (*fine-grain parallelism*). The suitable approach depends

on the given hardware and software environment, e.g. latency and bandwith of the communication network or availability of thin low-level APIs.

Dynamic load balancing: This aspect is closely connected to the previous topic. Under certain conditions it may be impossible or unpredictable to split a problem into a set of evenly sized tasks. A suitable mechanism for *dynamic* load balancing is required to keep *all* compute resources as busy as possible. The load balancing problem becomes even more prominent if the (virtual) parallel computer system consists of compute nodes with different CPU performances (e.g. heterogeneous resources in network of workstations).

Replicated data model: Storing all data in a replicated fashion in memory on all nodes is the first approach to implement task parallelism. Its advantage is that in general less communication is required, but this programming approach shows problem size limitations.

Distributed data model: To offer solutions for *large* problems where per node memory resources are not sufficient, *data distribution* is the only way to tackle such *grand challenge* problems. It is desirable that the total memory of a parallel computer system can be allocated to keep large arrays. Then, the largest problem which can be handled scales with the number of compute nodes involved in the calculation. Large arrays like **F** and/or **P** are limiting actual realizations of simulations in quantum chemistry. These arrays need to be distributed.
A user-friendly support of the distributed data model is *not* provided in common message-passing libaries (MPI, PVM), and thus, it is the objective of software projects discussed in this tutorial.

Interface to numerical libraries: A high-level interface to common numerical libraries (e.g. ScaLAPACK) should be available. Desirable are routines for matrix operations of distributed arrays (matrix multiply) as well as an interface to parallel eigensolvers[a] .

One-sided communication: In a programming model supporting distributed data, a "lightweighted" access to remote data is important. Usual point-to-point connections implemented in message-passing environments require the interaction of CPU resources on both sides. Tools described in this tutorial provide methods to access remote data without interferring with application process running on a remote CPU.

A common classification scheme for communication operations considers to what extent compute nodes are involved:

Collective operations: These are operations where *all* computing nodes of a given set are involved. Typically, they distribute work across the nodes. Examples are global summation, broadcast, and synchronization operations.

[a] for details see contribution given by I. Gutheil[1] in this volume

Non-collective operations: These type of communication operations fall into synchronous and one-sided (or asynchronous) operations:

Synchronous operations: In common message passing environments (e.g. MPI) any point-to-point data transfer consists of a complementary pair of *send/receive* invocations. On both sides CPU resources are allocated to perform the data transfer. In this type of communication an implicit pairwise synchronization is involved.

One-sided operations: Within these operations only one node is active. Communication or data transfer is performed without interferring with the other node which keeps remote data in memory. Examples are one-sided read and write operations from/into remote memory or atomic update operations on remote locations.

Atomic operations: To update the contents of variables in a save way it is necessary to prevent concurrent read or write operations on the same memory locations. This is achieved by locking the critical code region, i.e. only *one* request is permitted to perform read/write actions. More complex atomic operations like *read-and-increment* or *accumulate* includes a locking mechanism in a transparent way.

3 Parallel Tools in Today's Production Codes

There are several program packages available to perform quantum chemical calculations of large molecules on massively-parallel platforms. Table 1 presents some important examples of programming tools developed in this context. For the sake of completeness, commonly used tools on shared-memory platforms are included[b] too.

The Global Array Toolkit[4,5] (GA) is the *de-facto* standard in the quantum chemistry community. The suite of tools in GA is primarily designed as middleware for quantum chemical program packages. Major application examples are those which are developed and maintained at PNNL like NWChem[9] and COLUMBUS[10] as well as GAMESS-UK[11], and applications coming from other fields. Beside providing a portable interface to global shared arrays the GA includes tools for dynamic memory allocation (MA), one-sided communication (AMRCI), and efficient I/O capablities for parallel I/O (e.g. ChemIO).

The Distributed Data Interface[13] (DDI) has been written for the GAMESS[12] (US) package, and provides a one-sided data access via communication with data-servers on top of MPI or TCP/IP sockets. Within the DFT code DGauss[15] a proprietary Distributed Matrix[16] library is used. A quite different programming approach is used in Gaussian[17] and MPQC[18]. In Gaussian, the parallelization is implemented using the LINDA programming model. The MPQC package follows an object-oriented design and parallelization is implemented by using the Scientific Computing Toolkit[19] C++ class library.

[b] for details regarding OpenMP see contribution given by M. Gerndt[2] in this volume

Table 1. Tools used by various quantum chemistry packages on multi-processor systems

Memory Architecture	Tool (Communication Lib)	Packages (examples)
distributed memory	Global Array Toolkit (TCGMSG, MPI, SHMEM, sockets)	NWChem, GAMESS-UK, COLUMBUS
	Distributed Data Interface (MPI, sockets)	GAMESS (US)
	Distributed Matrix Lib (SHMEM, PVM, MPI)	DGauss
	LINDA (MPI, SHMEM)	Gaussian
	Scientific Computing Toolkit (PVM, MPI, NX, IPC)	MPQC
shared memory	OpenMP Microtasking	MOLPRO, Gaussian DGauss

4 The TCGMSG Library

In the late 80s and early 90s portable message passing libraries like PVM or MPI did not exists yet. To provide a compact, user-friendly, and portable programmer's interface for message passing R. J. Harrison implemented the TCGMSG[3] (Theoretical chemistry group message-passing toolkit) toolkit. The programming model and interface is directly modelled after PARMACS[22] developed at Argonne National Lab.

In the UNIX environment communication is done via TCP sockets. If identical processes are running on shared memory machine the faster communication mechanism provided by the hardware is used. Thus, applications can be built to run on an entire network of machines with local communication running at memory bandwidth and remote communication running at the corresponding network speed. On true message-passing machines TCGMSG is just a thin layer over the system interface (e.g. SHMEM on CRAY T3E).

Asynchronous communication is available in TCGMSG on machines supporting it. Otherwise, send operations block until the message is explicitly received, and messages from a particular process can only be received in the order sent. As far as buffering provided by the transport layer permits, messages are actually sent without any synchronization between sender and receiver.

TCGMSG supports FORTRAN and C bindings, Table 2 gives an overview of the TCGMSG API.

One remarkable feature is the `integer function NXTVAL(MPROC)` which implements a shared counter by communicating with a dedicated server process. It returns the next counter associated with a single active loop (0,1,2,...). `MPROC` is the number of processes actively requesting values. This `NXTVAL` function allows the implementation of dynamic load balancing within loops (work parallelism).

Table 2. Basic functions of TCGMSG

Operation	FORTRAN	C
Initialization	call PBEGINF()	PBEGIN_(...)
Termination	call PEND()	PEND_()
Identification		
Number of nodes	NNODES()	NNODES_()
Node ID	NODEID()	NODEID_()
Communication Operations		
Send	call SND(...)	SND_(...)
Receive	call RCV(...)	RCV_(...)
Broadcast	call BRDCST(...)	BRDCST_(...)
Synchronization	call SYNCH(...)	SYNCH_(...)
Shared counter	NXTVAL(...)	NXTVAL_(...)
Global operation	call DGOP(...)	DGOP_(...)
Utilities		
Print statistics	call STATS()	STATS_()
Wall time	TCGTIME()	TCGTIME_()

A FORTRAN example may look like the following code snippet:

```
   next = NXTVAL(MPROC)

   do i = 0, big
      if (i .eq. next) then

!     ... do work for iteration i

         next = NXTVAL(MPROC)
      endif
   end do
```

A complete code example showing the usage of the `NXTVAL` function can be found in Appendix A.

5 The Global Array Toolkit

5.1 Overview

The Global Array Toolkit[4] (GA) is written and maintained at the William R. Wiley Environmental Molecular Sciences Laboratory (EMSL) at Pacific Northwest National Laboratory (PNNL). The principal development work is done by J. Nieplocha, the latest version is release 3[5,6].

The GA provides a portable shared memory style programming environment which consists of a certain *programming model* as well as a *logical view* of a virtual machine (machine model)[20]. The virtual machine might be a massively-parallel distributed memory or a scalable shared memory platform. It is viewed as an ensemble of nodes consisting of CPU(s) and local memory (Figure 1). The whole

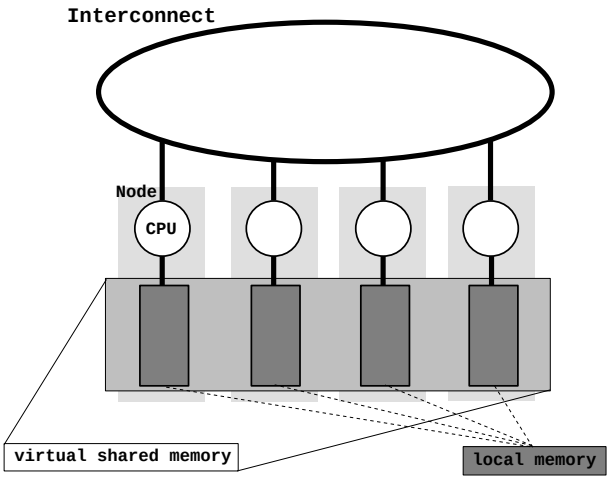

Figure 1. The Global Array machine model

memory is accessible via global arrays. These logically shared data are divided into local and remote portions, and the GA recognizes variable data transfer costs required to access the data (fast access to local data, slower access to remote data). In acknowledging the difference of local and remote data access, the GA model exposes to the programmer the *Non-Uniform Memory Access* (NUMA) architecture of computer systems found today.

5.2 Basic Features

The GA programming environment supports the development of parallel programs in C or/and FORTRAN. Main application areas are codes that operate on distributed dense multidimensional arrays and/or require shared memory programming style or one-sided communication. GA encapsulates many details of array distribution and addressing so that the programming effort is substantially reduced.

The basic shared memory operations supported include the one-sided communication operations *get*, *put*, *scatter*, and *gather*. The atomic operations *accumulate* and *read-and-increment* complete the list of fundamental functions. These operations are truely *one-sided* and will complete regardless of actions taken by a remote process that owns the data.

Details of the data distribution, addressing, and communication are encapsulated in the GA objects (internal data structures). The information on actual distribution patterns and locality of data can be obtained by library routines. Advantage of this information is taken whenever data locality is important.

At the user level, transfer operations of arrays or array patches are initiated using an array index interface rather than addresses. GA does not require the user to specify process IDs for accessing remote data. Any global array index-to-address translation and estimation of target IDs is done internally.

The GA is not built on top of any particular message-passing library but it

requires one to initialize the parallel environment and create processes. The one-sided communication required to support a shared memory programming model is facilitated by the ARMCI library included in the GA package[c] .

5.3 Using the GA Toolkit

Supported Plattforms

The current GA release supports *homogeneous* hardware platforms as follows:

- massively-parallel processor (MPP) systems (CRAY T3E, IBM SP)
- scalable shared memory systems (CRAY PVP, SGI Origin, Fujitsu VX/VPP)
- cluster of workstations (all UNIX flavours including Linux)
- SMP server (UNIX)
- multi-processor NT server

Selection of message-passing library

GA is working on top of a message-passing library which is selected at compile time on a given platform. For example, the message-passing library is used on some platforms to fork processes, implement broadcast or global operations (`ga_dgop`). Currently, either TCGMSG (s. section 4) or MPI can be used as an interface. Since TCGMSG is only a small set of routines and provides a convenient interface it is used per default on UNIX systems. On NT server the GA is build on top of WMPI, a NT implementation derived from MPICH.

Dependency on other software

Besides the message-passing interface there are additional software tools which are required by GA:

- MA (Memory Allocator) by Greg Thomas, a library for portable memory management;
- ARMCI, a one-sided communication library used by GA as its run-time system;
- BLAS library is required for the eigensolver and matrix multiply (`ga_dgemm`);
- GA eigensolver, `ga_diag`, is a wrapper for the eigensolver from PeIGS[8,d];
- LAPACK library is required for the eigensolver;
- MPI, SCALAPACK, PBBLAS, and BLACS libraries are required for some linear algebra functions (`ga_lu_solve`, `ga_cholesky`, `ga_llt_solve`, `ga_spd_invert`, `ga_solve`);

[c] In earlier versions of GA, the one-sided communication was implemented directly inside GA.
[d] application of PeIGS and performance issues are discussed by I. Gutheil[1]

5.4 Getting started with GA

Table 3 gives an overview about the most important functions implemented in GA, Table 4 summarizes some of the utility operations.

Table 3. Basic functionality provided by GA (only 2D array operations on whole arrays are listed)

Operation	FORTRAN	C
Initialization	ga_initialize()	GA_Initialize()
Termination	ga_terminate()	GA_Terminate()
Creation of arrays		
Regular dist.	ga_create(...)	NGA_Create(...)
Irregular dist.	ga_create_irreg(...)	NGA_Create_irreg(...)
Duplication	ga_duplicate(...)	GA_Duplicate(...)
Destroying	ga_destroy(...)	GA_Destroy(...)
One-sided operations		
Put	ga_put(...)	NGA_Put(...)
Get	ga_get(...)	NGA_Get(...)
Atomic accumulate	ga_acc(...)	NGA_Acc(...)
Atomic read and increment	ga_read_inc(...)	NGA_Read_inc(...)
Scatter	ga_scatter(...)	NGA_Scatter(...)
Gather	ga_gather(...)	NGA_Gather(...)
Interprocess synchronization		
Lock	ga_lock(...)	GA_lock(...)
Unlock	ga_unlock(...)	GA_unlock(...)
Fence	ga_init_fence()	GA_Init_fence()
	ga_fence()	GA_Fence()
Barrier	ga_sync()	GA_Sync()
Collective array operations		
Basic array operations		
Init zero	ga_zero(...)	GA_Zero(...)
Fill	ga_fill(...)	GA_Fill(...)
Scale	ga_scale(...)	GA_Scale(...)
Copy	ga_copy(...)	GA_Copy(...)
Linear algebra		
Matrix add	ga_add(...)	GA_Add(...)
Matrix multiply	ga_dgemm(...)	GA_Dgemm(...)
Dot product	ga_ddot(...)	GA_Ddot(...)
Symmetrization	ga_symmetrize(...)	GA_Symmetrize(...)
Transposition	ga_transpose(...)	GA_Transpose(...)

Table 4. Utility operations in GA

Operation	FORTRAN	C
Locality of data		
Locate	ga_locate(...)	NGA_Locate(...)
Find distrib.	ga_distribute(...)	NGA_Distribute(...)
Accessing	ga_acess(...)	NGA_Access(...)
Process information		
Process ID	ga_nodeid()	GA_Nodeid()
Nodes	ga_nnodes()	GA_Nnodes()
Memory availability		
Available memory	ga_memory_avail()	GA_Memory_avail()
Used memory	ga_inquire_memory()	GA_Inquire_memory()
Wrappers to reduction/broadcast operations		
Broadcast	ga_brdcst(...)	GA_brdcst(...)
Global operation	ga_dgop(...)	GA_Dgop(...)
	ga_igop(...)	GA_Igop(...)
Print detailed informations		
Print array	ga_print(...)	GA_Print(...)
Statistics	ga_print_stats()	GA_Print_stats()

The principal structure of a GA program should look like:

- when GA runs on top of MPI

```
   FORTRAN                      C

call mpi_init(...)          MPI_Init(...)       ! start MPI
call ga_initialize()        GA_Initialize()     ! start GA
status = ma_init(...)       MA_Init(...)        ! start MA

 .... do some work          ... do some work

call ga_terminate()         GA_Terminate()      ! tidy up GA
call mpi_finalize()         MPI_Finalize()      ! tidy up MPI
stop                        exit()              ! exit
```

- when GA runs on top of TCGMSG

```
   FORTRAN                      C

call pbeginf()              PBEGIN_(...)        ! start TCGMSG
call ga_initialize()        GA_Initialize()     ! start GA
status = ma_init(...)       MA_Init(...)        ! start MA

 .... do some work          ... do some work

call ga_terminate()         GA_Terminate()      ! tidy up GA
call pend()                 PEND_()             ! tidy up TCGMSG
stop                        exit()              ! exit
```

Examples: The Parallel Process Environment

The following two examples will demonstrate the first basic steps to employ the GA toolkit. In the first example the parallel process environment is provided and each node prints its unique node ID.

```
program example1
implicit none
include 'global.fh'

integer         :: n_nodes, me
logical         :: stat

call pbeginf()                                          ! init of TCGMSG
call ga_initialize()                                    ! init GA
n_nodes = ga_nnodes()                                   ! get number of nodes
me      = ga_nodeid()                                   ! who am i
print *, 'Hi, Iam node ', me, ' of ', n_nodes, 'nodes'

call ga_terminate()                                     ! tidy up GA
call pend()                                             ! tidy up TCGMSG
stop
end program example1
```

If the program is running on 4 nodes it generates the following output:

```
Hi, Iam node  3  of  4 nodes
Hi, Iam node  0  of  4 nodes
Hi, Iam node  1  of  4 nodes
Hi, Iam node  2  of  4 nodes
```

Beginning with release 3, the node numbering in GA and TCGMSG is coherent, e.g. a code snippet like the following reports identical node IDs in a TCGMSG and GA environment.

```
tcg_n_nodes = NNODES ()          ! TCGMSG nodes cntr
tcg_me      = NODEID ()          ! TCGMSG nodeid
ga_n_nodes  = ga_nnodes()        ! GA nodes cntr
ga_me       = ga_nodeid()        ! GA nodeid

print *, 'TCGMSG: #nodes:', tcg_n_nodes, ' on node', tcg_me
print *, '    GA: #nodes:', ga_n_nodes , ' on node', ga_me
```

Example: Creating a Regular Global 2D Array

The next example introduces the creation of a 2D array. The syntax of the corresponding GA function `ga_create` is as follows:

```
logical function ga_create(type, dim1, dim2, name, chunk1, chunk2, g_a)
        integer       type      - MA type                              [input]
        integer       dim1/2    - array(dim1,dim2) as in FORTRAN       [input]
        character*(*) name      - a unique character string            [input]
        integer       chunk1/2  - minimum size that dimensions should
                                  be chunked up into                   [input]
        integer       g_a       - handle for future references         [output]
```

Setting `chunk1 = dim1` gives distribution by vertical strips (`chunk2*columns`); setting `chunk2 = dim2` gives distribution by horizontal strips (`chunk1*rows`). Actual chunks will be modified so that they are at least the size of the minimum and each process has either zero or one chunk. Specifying `chunk1/2` as $\langle$ 1 will cause that dimension to be distributed evenly. `ga_create` is a *collective* operation.

The following code snippet demonstrates the creation of a regular 2D array. Assuming that the code is started on `4` nodes the distribution of the squared `100x100` array named `A` is done by horizontal strips. Figure 2 shows various distribution patterns depending on the chunk sizes in the `ga_create` call. (In the example below the data type `MT_DBL` means double-precision).

```
status = ma_init(...)                                  ! allocate memory
call ga_initialize()                                   ! init GA
status = ga_create ( MT_DBL, 100, 100, 'A', 25, 100, ga_hndl)
call ga_destroy ( ga_hndl )
call ga_terminate()
```

Complete code examples demonstrating the creation of a global regular and irregular 2D arrays, the usage of matrix multiply interface and some utility functions are listed in Appendices B-F.

5.5 About Performance: Matrix Multiply

To give some impression about the performance we present some numbers for matrix multiply operations obtained on our CRAY T3E-1200. Three matrix multiply routines are compared: the optimized BLAS routine `SGEMM` (64 bit, running on a single node), the parallel `GA_DGEMM` provided by GA (code example Appendix E), and a matrix multiply version implemented in High Performance Fortran (HPF).

Fig. 3 shows the single node performance for a $N = 1000$ matrix multiply. The performance degradation for `GA_DGEMM` is small, and is mainly due to the array index-to-address conversions. More in detail: using the native `SGEMM` routine one obtains 592 MFlop/s whereas `GA_GEMM` achieves 563 MFlop/s. Please note the poor performance of 6 MFlop/s for the HPF code. The dependence of the global performance on the number of nodes for a fixed problem size is showing in Fig. 4. For the given problem size $N = 1000$, the efficiency of `GA_DGEMM` ranges from 0.92 to 0.57 for two up to eight nodes, respectively. If one measures the performance with respect to the problem size on eight nodes (s. Fig. 5) one can show that the dimension of matrices should be at least in the order of some hundreds to obtain a reasonable speed on CRAY T3E.

5.6 Further components of the Global Array toolkit

The *Memory Allocator* (MA) is a library of routines that comprises a dynamic memory allocator for use by C, FORTRAN, or mixed-language applications. MA is designed to be portable across a variety of platforms. C applications can benefit from using MA instead of the ordinary `malloc()` and `free()` routines because of the extra features MA provides: both heap and stack memory management disciplines, debugging and verification support, usage statistics, and quantitative

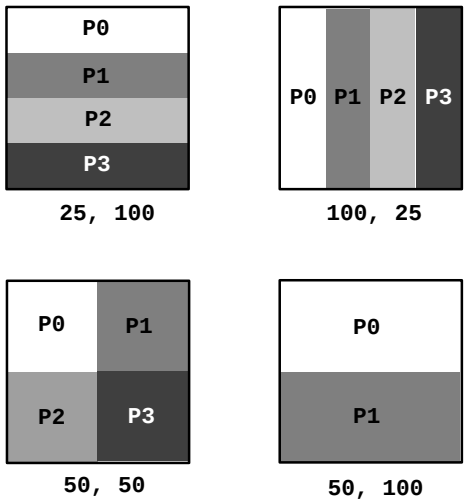

Figure 2. Creation of global 2D arrays: distribution patterns for various chunk sizes i, j

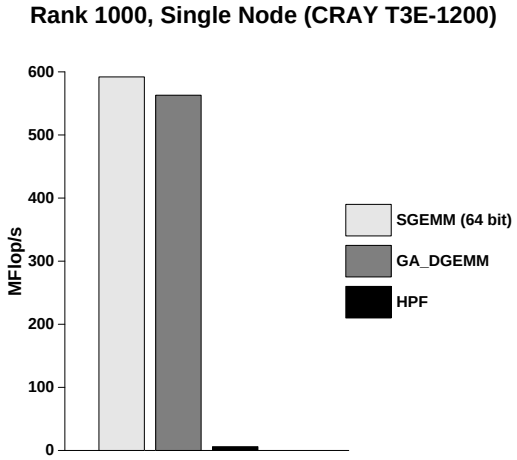

Figure 3. Matrix Multiply: Single Node Performance (CRAY T3E-1200)

memory availability information. FORTRAN applications can take advantage of the same features, and predecessors of FORTRAN 9x codes may in fact require a library such as MA because dynamic memory allocation is not supported. One important advantage of MA should be noted: memory leaks are avoided due to the management strategies of MA.

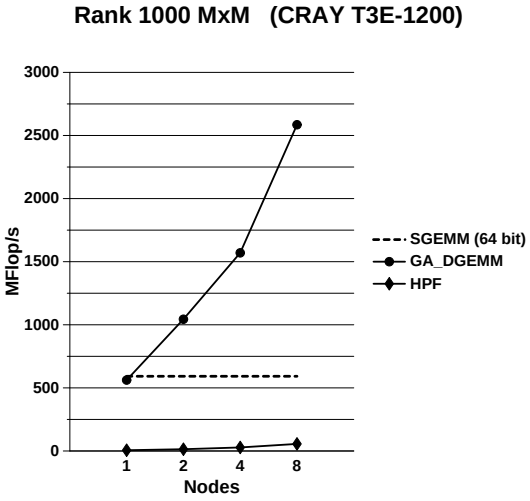

Figure 4. Matrix Multiply: Performance Scaling vs. Number of Nodes (CRAY T3E-1200)

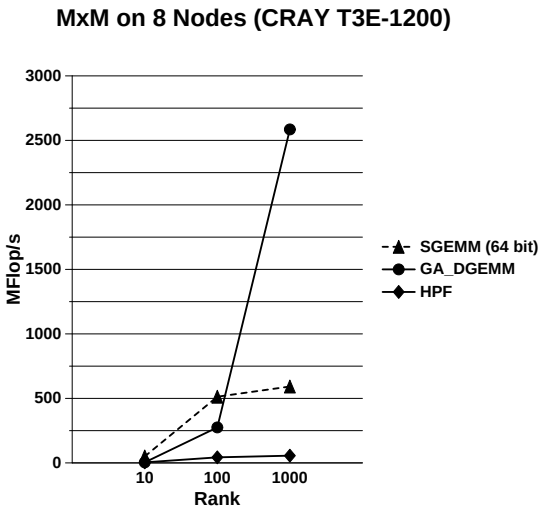

Figure 5. Matrix Multiply: Performance Scaling vs. Problem Size (CRAY T3E-1200)

A suite of independent parallel I/O libraries for high-performance computers building a chemistry I/O API, ChemIO[23], has been developed:

- Disk Resident Arrays (DRA),
- Shared Files (SF), and
- Exclusive Access Files (EAF).

All the components have an asynchronous API allowing the applications to overlap expensive I/O with computations. As an example, the DRA layer is employed in the RI-MP2 module in NWChem, thus some aspects of the DRA are discussed in the following.

Disk Resident Arrays (DRA) extend the Global Arrays (GA) programming model to disk. The library encapsulates the details of data layout, addressing and I/O transfer in disk arrays objects. Disk resident arrays resemble global arrays except that they reside on disk instead of in main memory. The main features of this model are:

- Data can be transferred between disk and global memory.
- I/O operations have a nonblocking interface to allow overlapping of I/O with computations.
- All I/O operations are collective.
- Either whole or sections of global arrays can be transferred between GA memory and the disk.

5.7 Guidelines for using GA

Some guidelines regarding a suitable application scenario for the GA toolkit may be summarized as follows[6]. Utilization of GA is preferred:

- for applications with dynamic and irregular communication patterns,
- for calculations gaining advantage from dynamic load balancing,
- if one-sided access to shared data is required,
- when coding in message-passing style is too complicated,
- if data locality is important.

Alternatives to GA – MPI, HPF, or OpenMP – might be considered,

- if there are regular communication patterns or only a few communication paths (e.g. nearest neighbour) often found in domain decomposition approaches,
- when synchronization after point-to-point message passing is needed, or
- if compiler parallelization is more effective e.g. OpenMP on shared memory platforms.

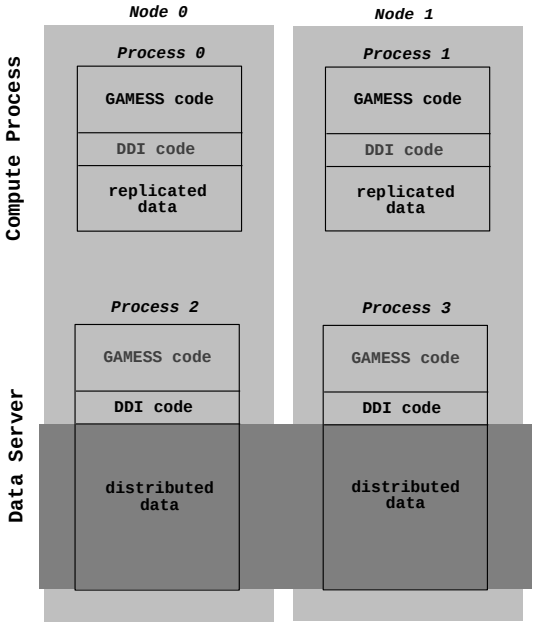

Figure 6. The Data Server model in DDI for two nodes (less active program components are shown in gray)

6 The Distributed Data Interface Used in GAMESS (US)

The Distributed Data Interface (DDI) was introduced in the public GAMESS[12] (US) in June 1999, and replaces the TCGMSG API used since 1991. The low level transport layers which DDI relies on are SHMEM, MPI-1, or TCP/IP sockets.

The DDI is similar to the GA concept in the sense that DDI attempts to exploit the entire machines memory as global shared memory. The implementation follows the concept of providing one-sided communication calls. There are three subroutine calls to access memory on remote nodes: `DDI_PUT`, `DDI_GET`, and `DDI_ACCUMULATE`.

At the present time, the DDI routines support only two dimensional FORTRAN arrays, organized in such a way that columns are kept on a single node's memory. Up to 10 matrices may be distributed in this fashion.

6.1 The Data Server concept

Since MPI-2 is still unavailable on most platforms, and the SHMEM[24] [e] is or will be available on a limited number of computer systems only, a data server model is implemented in DDI on other platforms (Figure 6).

On each node two processes are running: one is doing the chemistry work (compute process) and the other one acts as data server supporting DDI *get*, *put*, and

[e] The logically shared, distributed memory access (SHMEM) routines provide low-latency, high-bandwidth communication.

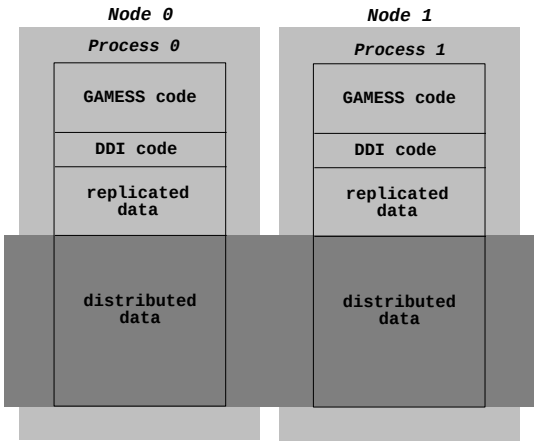

Figure 7. The DDI implementation on CRAY T3E for two nodes

Table 5. Some useful DDI Functions

Operation	Functions
Global Operations	
Initialization	DDI_PBEG(...)
Termination	DDI_PEND(...)
Number of nodes, ID	DDI_NPROC(...)
Allocate shared memory	DDI_MEMORY(...)
Create distrib. matrix	DDI_CREATE(...)
Destroy matrix	DDI_DESTROY(...)
Synchronization	DDI_SYNC(...)
Global sum (fp)	DDI_GSUMF(...)
Broadcast	DDI_BCAST(...)
Point-to-Point Operations	
Synchronous send	DDI_SEND(...)
Synchronous receive	DDI_RECV(...)
Get next index	DDI_DLBNEXT(...)
Get block of matrix	DDI_GET(...)
Put block of matrix	DDI_PUT(...)
Accumulate data into block	DDI_ACC(...)

accumulate requests. Interupts are handled by the operating system, as the data servers are distinct processes.

On CRAY MPP systems an efficient, hardware supported API for (fast) one-sided communication is provided (SHMEM). Therefore, the data server model can be dropped which allows for a more compact implementation. The memory image of a GAMESS process on T3E is shown in Figure 7.

DDI provides an interface for ordinary message passing parallel programming. A summary of important operations can be found in Table 5. There are functions to initialize and terminate the various processes, point-to-point send and receive, and collective operations like global sum and the broadcast. In addition, functions for distributed data manipulation, which include creation and destruction of the arrays, the put, get, and accumulation operations mentioned above, are available.

Note, that the current version of DDI is not intended to be a general parallel programming library, e.g. there are no portable memory management routines included. Thus, parts of the GAMESS distribution are still required.

6.2 DDI Example

To illustrate the DDI a very simple example[14] is given below. It shows the proper initialization and closure of the DDI library, requests replicated memory but not distributed memory.

```
  program bcast
  implicit double precision(a-h,o-z)
  parameter (maxmsg=500000)
  common /fmcom / xx(1)
  data exetyp/8hRUN     /

  nwdvar=2                              ! open DDI, tell it integer  ...
  call ddi_pbeg(nwdvar)                 ! ... word length is 32 bit
  memrep=maxmsg                         ! request allocation of only
  memddi=0                              ! replicated memory
  call ddi_memory(memrep,memddi,exetyp)
  call setfm(memrep)
  call ddi_nproc(nproc,me)              ! who am I
  master=0
  if(me.eq.master) print *, 'running ', nproc, ' processes'
  call valfm(loadfm)                    ! allocate a replicated array
  lbuff = loadfm + 1
  last  = lbuff  + maxmsg
  need = last - loadfm - 1
  call getfm(need)

  if (me.eq.master) then
     do i=1,maxmsg                      ! fill it up with ones
        xx(lbuff-1+i) = 1.0d+00
     end do
  end if

!                                 send it to all the other compute processes
  call ddi_bcast(102,'F',xx(lbuff),maxmsg,master)

  call retfm(need)                      ! replicated storage done
  istat=0
  call ddi_pend(istat)                  ! close the DDI library gracefully
  stop
  end
```

7 Further Reading

Students which are not necessarily familiar with parallel computational chemistry are encouraged to study the presentation *Parallel Computing in Chemistry* by Roland Lindh[21]. Access to further documentation regarding the Global Array Tookit can be found on the Web[5]. The *Global Array User's Manual* [6] explains most of the details required to implement parallel software using the GA. An excellent introduction to DDI as well as to the global shared memory programming model is given in the GAMESS (US) documentation[14]. Some of the ideas for the Global Array presentation were "borrowed" from the on-line slide-show *Global Arrays* by Krister Dackland[20].

8 Summary

Driven by practical requirements in the field of quantum chemistry, stable and robust tools for parallel programming were and are developed. The TCGMSG library was (and is) *the* "quantum chemist's" message-passing interface. Besides other tools and libraries, the Global Array Toolkit (GA) can be viewed as the *de-facto* standard for parallel applications in the quantum chemistry community. Using the GA one benefits from it since the Global Array Toolkit

- provides a shared memory programming style ...
- ... on distributed memory *and* true shared memory architectures,
- supports an efficient NUMA,
- is user-friendly, and
- is open-source.

Due to performance reasons, HPF based applications play only a minor role in the quantum-chemistry arena.

References

1. I. Gutheil, *Basic numerical libraries for parallel systems*, Modern Methods and Algorithms of Quantum Chemistry, Winterschool at Forschungszentrum Jülich, 21-25 February 2000 (this volume).
2. M. Gerndt, *Parallel programming models, tools and performance analysis*, Modern Methods and Algorithms of Quantum Chemistry, Winterschool at Forschungszentrum Jülich, 21-25 February 2000 (this volume).
3. R.J. Harrison, The TCGMSG Message-Passing Toolkit, Pacific Northwest National Laboratory, version 4.04, 1994.
   `http://www.emsl.pnl.gov:2080/docs/nwchem/tcgmsg.html`
4. J. Nieplocha, R.J. Harrison, R.J. Littlefield, Global Arrays: A portable "shared-memory" programming model for distributed memory computers, Proc. Supercomputing'94 , 340-349 (1994).

J. Nieplocha, R.J. Harrison, R.J. Littlefield, The Global Array programming model for high performance scientific computing, SIAM-News **September**, (1995).
J. Nieplocha, R.J. Harrison, R.J. Littlefield, Global Arrays: A nonuniform memory access programming model for high-performance computers, The Journal of Supercomputing **10**, 197-220 (1996).

5. J. Nieplocha, j_nieplocha@pnl.gov, Environmental Molecular Sciences Laboratory, Pacific Northwest National Laboratory.
   http://www.emsl.pnl.gov:2080/docs/global/ga.html
6. J. Nieplocha, J. Ju, Global Arrays User's Manual, Pacific Northwest National Laboratory, 1999.
7. further information: http://www.emsl.pnl.gov:2080/docs/parsoft/armci
8. please contact George Fann gi_fann@pnl.gov about PeIGS.
9. Northwest Computational Chemistry Package (NWChem) 3.3, Anchell, J.; Apra, E.; Bernholdt, D.; Borowski, P.; Clark, T.; Clerc, D.; Dachsel, H.; Deegan, M.; Dupuis, M.; Dyall, K.; Fann, G.; Früchtl, H.; Gutowski, M.; Harrison, R.; Hess, A.; Jaffe, J.; Kendall, R.; Kobayashi, R.; Kutteh, R.; Lin, Z.; Littlefield, R.; Long, X.; Meng, B.; Nichols, J.; Nieplocha, J.; Rendall, A.; Stave, M.; Straatsma, T.; Taylor, H.; Thomas, G.; Wolinski, K.; Wong, A.; "NWChem, A Computational Chemistry Package for Parallel Computers, Version 3.2.1" (1998), Pacific Northwest National Laboratory, Richland, Washington 99352-0999, USA.
   D.E. Bernholdt, E. Apra, H.A. Früchtl, M.F. Guest, R.J. Harrison, R.A. Kendall, R.A. Kutteh, X. Long, J.B. Nicholas, J.A. Nichols, H.L. Taylor, A.T. Wong, G.I. Fann, R.J. Littlefield and J. Nieplocha, Parallel Computational Chemistry Made Easier: The Development of NWChem, Int. J. Quantum Chem. Symposium **29**, 475-483 (1995).
   http://www.emsl.pnl.gov:2080/docs/nwchem/
10. COLUMBUS, H. Lischka, R. Shepard, I. Shavitt, F.B. Brown, R.M. Pitzer, R. Ahlrichs, H.-J. Böhm, C. Ehrhardt, R. Gdanitz, P. Scharf, H. Schiffer, M. Schindler, D.C. Comeau, M. Pepper, K. Kim, P.G. Szalay, J.-G. Zhao, E. Stahlberg, G. Kedziora, G. Gawboy, H. Dachsel, S. Irle, M. Dallos, Th. Müller, T. Kovar, M. Ernzerhof, A.H.H. Chang, V. Parasuk, M. Schüler, P. Höchtl.
    http://www.itc.univie.ac.at/~hans/Columbus/columbus_parallel.html
11. GAMESS-UK is a package of ab initio programs written by M.F. Guest, J.H. van Lenthe, J. Kendrick, K. Schöffel, and P. Sherwood, with contributions from R.D. Amos, R.J. Buenker, H.J.J. van Dam, M. Dupuis, N.C. Handy, I.H. Hillier, P.J. Knowles, V. Bonacic-Koutecky, W. von Niessen, R.J. Harrison, A.P. Rendell, V.R. Saunders, A.J. Stone and A.H. de Vries. The package is derived from the original GAMESS code due to M. Dupuis, D. Spangler and J. Wendoloski, NRCC Software Catalog, Vol. 1, Program No. QG01 (GAMESS), 1980.
    GAMESS 6.2, M.F. Guest, J. Kendrick, J.H. van Lenthe and P. Sherwood, Computing for Science Ltd., 1999.
    http://wserv1.dl.ac.uk/CFS/

12. GAMESS rel. June 1999, M.W. Schmidt, K.K. Baldridge, J.A. Boatz, S.T. Elbert, M.S. Gordon, J.H. Jensen, S. Koseki, N. Matsunaga, K.A. Nguyen, S.J. Su, T.L. Windus, M. Dupuis, J.A. Montgomery, J. Comput. Chem. **14**, 1347-1363 (1993).
`http://www.msg.ameslab.gov/GAMESS/GAMESS.html`
13. G.D. Fletcher, M.W. Schmidt, Distributed Data Interface - "SPMD" Data Server Model, 1999.
14. Programmer's Reference, GAMESS, June 1999.
15. DGauss 4.1, Oxford Molecular, 1998-1999.
`http://www.oxmol.com/software/unichem/dgauss`
16. E.A. Stahlberg, private communication, 1998.
17. Gaussian 98, Revision A.7, M.J. Frisch, G.W. Trucks, H.B. Schlegel, G.E. Scuseria, M.A. Robb, J.R. Cheeseman, V.G. Zakrzewski, J.A. Montgomery, Jr., R.E. Stratmann, J.C. Burant, S. Dapprich, J.M. Millam, A.D. Daniels, K.N. Kudin, M.C. Strain, O. Farkas, J. Tomasi, V. Barone, M. Cossi, R. Cammi, B. Mennucci, C. Pomelli, C. Adamo, S. Clifford, J. Ochterski, G.A. Petersson, P.Y. Ayala, Q. Cui, K. Morokuma, D.K. Malick, A.D. Rabuck, K. Raghavachari, J.B. Foresman, J. Cioslowski, J.V. Ortiz, A.G. Baboul, B.B. Stefanov, G. Liu, A. Liashenko, P. Piskorz, I. Komaromi, R. Gomperts, R.L. Martin, D.J. Fox, T. Keith, M.A. Al-Laham, C.Y. Peng, A. Nanayakkara, C. Gonzalez, M. Challacombe, P.M.W. Gill, B. Johnson, W. Chen, M.W. Wong, J.L. Andres, C. Gonzalez, M. Head-Gordon, E.S. Replogle, and J.A. Pople, Gaussian, Inc., Pittsburgh PA, 1998, `www.gaussian.com`
18. MPQC (Massively Parallel Quantum Chemistry) Limit Point Systems, Inc., Fremont, CA, 1999.
`http://aros.ca.sandia.gov/~cljanss/mpqc/`
19. C.L. Janssen, E.T. Seidl, and M.E. Colvin, *Object Oriented Implementation of Parallel Ab Initio Programs*, ACS Symposium Series 592, *Parallel Computers in Computational Chemistry*, T. Mattson, Ed., 1995.
`http://aros.ca.sandia.gov/ cljanss/sc/`
20. K. Dackland, *Global Array Toolkit*, slide show, 1999.
`http://www.hpc2n.umu.se/events/courses/slides/GA/index.htm`
21. R. Lindh, *Parallel Computing in Chemistry*.
`http://www.hpc2n.umu.se/events/ngssc/98/parallel/notes/lectures_26-27/`
22. J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, R. Stevens. *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, Inc., 1987.
23. `http://www.emsl.pnl.gov:2080/docs/parsoft/chemio/chemio.html`
24. CRAY T3E Fortran Optimization Guide, SG-2518 3.0, Cray Res. Inc., 1997.

Please note that the following examples are written for tutorial purposes only, i.e. the author acknowledges that there exists more elegant and sophisticated solutions.

Appendix A: Shared Counter using TCGMSG NXTVAL Function

Program Code

```
program sc
implicit none

integer, parameter      :: NTASKS = 20
integer                 :: nodes, me
integer, external       :: NNODES, NODEID, NXTVAL
integer                 :: next, i, ntsks
logical                 :: master

nodes  = NNODES()
me     = NODEID()
master = me == 0

if ( master ) print *, 'run on', nodes, ' nodes'

ntsks = 0
next = NXTVAL ( nodes )

do i=0, NTASKS-1
   if ( next == i ) then
      print *, 'node', me, ' do task', next

      call work ( 5. )

      ntsks = ntsks + 1
      next = NXTVAL ( nodes )
   end if
end do

print *, 'node', me, ' processed', ntsks, ' tasks'

end program sc

!------------------------------------------------------------------------------

subroutine work ( fctr )
implicit none
real            :: fctr
real            :: tmp
integer         :: secs, left, error

call RANDOM_NUMBER ( tmp )
secs = fctr * tmp

call pxfsleep ( secs, left, error )             !sleep for <secs> seconds

return
end subroutine work
```

Example Output

```
 run on 6  nodes
 node 1  do task 3
 node 1  do task 7
 node 1  do task 14
 node 1  processed 3  tasks
 node 4  do task 2
 node 4  do task 9
 node 4  do task 15
 node 4  processed 3  tasks
...
 node 3  do task 0
 node 3  do task 6
 node 3  do task 13
 node 3  do task 18
 node 3  processed 4  tasks
 run on 6  nodes
 node 0  do task 5
 node 0  do task 8
 node 0  do task 12
 node 0  do task 19
 node 0  processed 4  tasks
```

Appendix B: Creation of Global 2D Array (I)

Program Code

```
program create_2d

implicit none

include 'mafdecls.fh'
include 'global.fh'

integer, parameter      :: MAXDIM = 100

integer         :: n_nodes, me, ga_hndl, owner
logical         :: status
integer         :: jnk_size, i

call pbeginf ()                                         ! init TCGMSG
call ga_initialize ()                                   ! init GA
n_nodes = ga_nnodes ()
me = ga_nodeid ()

jnk_size = MAXDIM/n_nodes

status = ma_init(MT_DBL, MAXDIM*jnk_size, MT_BYTE)     ! allocate memory
status = ga_create ( MT_DBL, MAXDIM, MAXDIM, 'A', jnk_size, MAXDIM, ga_hndl)

do i = 1, MAXDIM, MAXDIM/n_nodes
   if ( ga_locate( ga_hndl, i, 1, owner) ) then
      if ( me == owner ) &
         print *,'PE', ga_nodeid(), ' owns rows ', i, ' - ', i+MAXDIM/n_nodes-1
   end if
end do
```

```
call ga_sync ()
status = ga_destroy ( ga_hndl )
call ga_terminate()
call pend()
stop
end program create_2d
```

Example Output

```
 PE 3  owns rows  76  -  100
 PE 2  owns rows  51  -  75
 PE 0  owns rows  1  -  25
 PE 1  owns rows  26  -  50
```

Appendix C: Creation of Global 2D Array (II)

Program Code

```
program crea_2d

implicit none

include 'mafdecls.fh'
include 'global.fh'

integer, parameter      :: MAXDIM = 100

integer         :: n_nodes, me, ga_hndl, owner
logical         :: status
integer         :: jnk_size, j

call pbeginf ()                                         ! init TCGMSG
call ga_initialize ()                                   ! init GA
n_nodes = ga_nnodes ()
me = ga_nodeid ()

jnk_size = MAXDIM/n_nodes

status = ma_init(MT_DBL, MAXDIM*jnk_size, MT_BYTE)        ! allocate memory

status = ga_create ( MT_DBL, MAXDIM, MAXDIM, 'A', MAXDIM, jnk_size, ga_hndl)

if ( me == 0 ) call ga_summarize ( .true. )

call ga_sync()
status = ga_destroy ( ga_hndl )
call ga_terminate()
call pend()
stop
end program crea_2d
```

Example Output

```
 Summary of allocated global arrays
-----------------------------------
  array 0 => double precision A(100,100),  handle: -1000
    (1:100,1:25) -> 0
    (1:100,26:50) -> 1
    (1:100,51:75) -> 2
    (1:100,76:100) -> 3
```

Appendix D: Print a Distributed Array

Program Code Snippet

```
...
integer, parameter      :: MAXDIM = 16
logical                 :: status
integer                 :: ga_hndl, chunk_size
...
status = ga_create ( MT_DBL, MAXDIM, MAXDIM, 'A', MAXDIM, chunk_size, ga_hndl)

call ga_sync()
call ga_print ( ga_hndl )
...
```

Example Output

```
 global array: A[1:16,1:16],  handle: -1000

           1           2           3           4           5           6
      ----------- ----------- ----------- ----------- ----------- -----------
   1      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
   2      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
   3      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
   4      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
   5      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
   6      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
   7      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
   8      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
   9      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
  10      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
  11      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
  12      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
  13      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
  14      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
  15      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
  16      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000

           7           8           9          10          11          12
      ----------- ----------- ----------- ----------- ----------- -----------
   1      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
   2      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
   3      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
   4      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
   5      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
   6      0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
```

```
  7       0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
  8       0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
  9       0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
 10       0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
 11       0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
 12       0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
 13       0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
 14       0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
 15       0.00000     0.00000     0.00000     0.00000     0.00000     0.00000
 16       0.00000     0.00000     0.00000     0.00000     0.00000     0.00000

         13          14          15          16
     ----------- ----------- ----------- -----------
  1       0.00000     0.00000     0.00000     0.00000
  2       0.00000     0.00000     0.00000     0.00000
  3       0.00000     0.00000     0.00000     0.00000
  4       0.00000     0.00000     0.00000     0.00000
  5       0.00000     0.00000     0.00000     0.00000
  6       0.00000     0.00000     0.00000     0.00000
  7       0.00000     0.00000     0.00000     0.00000
  8       0.00000     0.00000     0.00000     0.00000
  9       0.00000     0.00000     0.00000     0.00000
 10       0.00000     0.00000     0.00000     0.00000
 11       0.00000     0.00000     0.00000     0.00000
 12       0.00000     0.00000     0.00000     0.00000
 13       0.00000     0.00000     0.00000     0.00000
 14       0.00000     0.00000     0.00000     0.00000
 15       0.00000     0.00000     0.00000     0.00000
 16       0.00000     0.00000     0.00000     0.00000
```

Appendix E: Parallel Matrix Multiply using GA_DGEMM

Program Code

```
program mxm_1

implicit none

include 'mafdecls.fh'
include 'global.fh'

integer                 :: n

integer                 :: n_nodes, me, hndl_a, hndl_b, hndl_c, owner
logical                 :: status
integer                 :: jnk_size, mem_size, j
logical                 :: master
real                    :: s, alpha, beta

real                    :: usr_time(2), usrt(2), mflops

call pbeginf ()                                         ! init TCGMSG
call ga_initialize ()                                   ! init GA
n_nodes = ga_nnodes ()
me = ga_nodeid ()
master = me == 0
```

```
call CPU_TIME ( usr_time(1) )

if ( master ) then
   print *, 'Nodes: ', n_nodes
   read (*,*) n
   print *, 'Rank = ', n
end if

call ga_sync()
call ga_brdcst ( MT_INT, n, MA_sizeof(MT_INT,1,MT_BYTE), 0 )

jnk_size = n/n_nodes

mem_size = MAX(6*jnk_size*jnk_size, 2000000)

status = ma_init(MT_DBL, 0 , mem_size)        ! allocate memory

call CPU_TIME ( usrt(1) )

status = ga_create ( MT_DBL, n, n, 'A', jnk_size, n, hndl_a)
status = ga_create ( MT_DBL, n, n, 'B', n, jnk_size, hndl_b)
status = ga_create ( MT_DBL, n, n, 'C', jnk_size, jnk_size, hndl_c)

s = 2.
call ga_fill_patch ( hndl_a, 1, n, 1, n, s )
s = 0.5
call ga_fill_patch ( hndl_b, 1, n, 1, n, s )
call ga_zero ( hndl_c )

call CPU_TIME ( usrt(2) )
if ( master ) then
   write (*, '(''usr time for init'',t24,f9.3,1x,''s'')') usrt(2)-usrt(1)
end if

call ga_sync()
call CPU_TIME ( usrt(1) )

call ga_dgemm( 'N',  'N', n, n, n, 1.0, hndl_a, hndl_b, 0.0, hndl_c )

call ga_sync()
call CPU_TIME ( usrt(2) )
usrt(1) = usrt(2)-usrt(1)
mflops=1.e-6*2*n*n*n/usrt(1)
if ( master ) then
   write (*, '(''usr time for matmul'',t24,f9.3,1x,''s'')') usrt(1)
   write (*, '(''global performance of'',t24,f9.3,1x,''mflop/s'')')  mflops
end if

if ( master ) then
   call ga_summarize ( .true. )
end if
call ga_sync()

status = ga_destroy ( hndl_c )
status = ga_destroy ( hndl_b )
status = ga_destroy ( hndl_c )
call ga_terminate()
call pend()
```

```
call CPU_TIME ( usr_time(2) )
if ( master ) then
   write (*, '(''total usr time'',t24,f9.3,1x,''s'')') usr_time(2)-usr_time(1)
end if

stop
end program mxm_1
```

Example Output

```
 Nodes:  16
 Rank =  1000
usr time for init          0.006 s
usr time for matmul        0.522 s
global performance of   3830.552 mflop/s
 Summary of allocated global arrays
-----------------------------------
  array 0 => double precision A(1000,1000),  handle: -1000
    (1:63,1:1000) -> 0
    (64:126,1:1000) -> 1
    (127:189,1:1000) -> 2
    (190:252,1:1000) -> 3
    (253:315,1:1000) -> 4
    (316:378,1:1000) -> 5
    (379:441,1:1000) -> 6
    (442:504,1:1000) -> 7
    (505:567,1:1000) -> 8
    (568:630,1:1000) -> 9
    (631:693,1:1000) -> 10
    (694:756,1:1000) -> 11
    (757:819,1:1000) -> 12
    (820:882,1:1000) -> 13
    (883:945,1:1000) -> 14
    (946:1000,1:1000) -> 15
  array 1 => double precision B(1000,1000),  handle: -999
    (1:1000,1:63) -> 0
    (1:1000,64:126) -> 1
    (1:1000,127:189) -> 2
    (1:1000,190:252) -> 3
    (1:1000,253:315) -> 4
    (1:1000,316:378) -> 5
    (1:1000,379:441) -> 6
    (1:1000,442:504) -> 7
    (1:1000,505:567) -> 8
    (1:1000,568:630) -> 9
    (1:1000,631:693) -> 10
    (1:1000,694:756) -> 11
    (1:1000,757:819) -> 12
    (1:1000,820:882) -> 13
    (1:1000,883:945) -> 14
    (1:1000,946:1000) -> 15
  array 2 => double precision C(1000,1000),  handle: -998
    (1:250,1:250) -> 0
    (251:500,1:250) -> 1
    (501:750,1:250) -> 2
    (751:1000,1:250) -> 3
    (1:250,251:500) -> 4
```

```
    (251:500,251:500) -> 5
    (501:750,251:500) -> 6
    (751:1000,251:500) -> 7
    (1:250,501:750) -> 8
    (251:500,501:750) -> 9
    (501:750,501:750) -> 10
    (751:1000,501:750) -> 11
    (1:250,751:1000) -> 12
    (251:500,751:1000) -> 13
    (501:750,751:1000) -> 14
    (751:1000,751:1000) -> 15
```

Appendix F: Irregular Distributed 2D Array

Program Code

```
program irreg_2d

implicit none

include 'mafdecls.fh'
include 'global.fh'

integer, parameter      :: MAXDIM = 8

integer                 :: n_nodes, me, ga_hndl, owner
logical                 :: status

integer, parameter      :: NBLOCK1 = 1, NBLOCK2 = 3
integer                 :: map1(NBLOCK1) = 1, map2(NBLOCK2) = (/1,2,5/)
integer                 :: node, mem_sze

logical                 :: master

call pbeginf ()                                         ! init TCGMSG
call ga_initialize ()                                   ! init GA
n_nodes = ga_nnodes ()
if ( n_nodes /= NBLOCK2 ) then
   print *, 'need', NBLOCK2, ' nodes, have', n_nodes, '!'
   stop 'wrong no. of nodes'
end if
me      = ga_nodeid ()
master  = me == 0

status = ma_init(MT_DBL, MAXDIM*MAXDIM, MT_BYTE)

status = ga_create_irreg ( MT_DBL, MAXDIM, MAXDIM, 'A', &
                           map1, NBLOCK1, map2, NBLOCK2, ga_hndl)

if ( master ) call ga_summarize ( .true. )

call ga_sync()
status = ga_destroy ( ga_hndl )
call ga_terminate()
call pend()
stop
end program irreg_2d
```

Example Output

```
 Summary of allocated global arrays
-----------------------------------
  array 0 => double precision A(8,8),  handle: -1000
    (1:8,1:1) -> 0
    (1:8,2:4) -> 1
    (1:8,5:8) -> 2
```